3rd USENIX Workshop on Hot Topics in Parallelism (HotPar '11)

Berkeley, CA
May 26–27, 2011

Day 1, Session 1

Summarized by Sabrina M. Neuman (sneuman@mit.edu)

Considerations When Evaluating Microprocessor Platforms

Michael Anderson, Bryan Catanzaro, Jike Chong, Ekaterina Gonina, Kurt Keutzer, Chao-Yue Lai, Mark Murphy, David Sheffield, Bor-Yiing Su, and Narayanan Sundaram, University of California, Berkeley

Bryan Catanzaro opened the HotPar '11 workshop with an examination of the problems plaguing GPU and CPU microprocessor platform comparisons. The two key conclusions of the investigation were that comparison results should be contextualized within a certain point of view, and that comparison results should be reproducible.

To illustrate the first conclusion, Catanzaro invoked the parable of the blind men and the elephant, where the men draw inconsistent conclusions because each collects data from a different point of view. He likened the men in the story to modern application researchers and architecture researchers, and suggested that comparison results need to be consistent with the point of view of their intended audience. The goals and values of application and architecture researchers were surveyed. Different points of view will require different sorts of comparisons, asserted Catanzaro. For example, large applications are useful for application researchers and micro-benchmarks are useful for architecture researchers.

To realize reproducible comparison results, as suggested by the second conclusion of the paper, Catanzaro argued that more detail must be presented with comparisons. Researchers should avoid making absolute claims about the superiority of one platform over another unless a full architectural study is performed. The structures of algorithms and data sets must be explained. The descriptions of the platforms being compared must be explicit. Catanzaro made a plea for researchers to practice good science by providing full details in their microprocessor platform comparisons, insisting that bad comparisons are holding back progress in the field.

Most questions centered on benchmarks as a means of comparison. Mike McCool (Intel) asked if there were any existing benchmarks that made for fair comparisons according to this work. Catanzaro replied that there are some good low-level benchmarks, but that micro-benchmarks are less useful than full applications. Dave Patterson (UC Berkeley) wondered if asking for reproducibility from cloud computing would be too restrictive, since it might require having to run on some particular cloud every time. Catanzaro replied that it would still be a good idea, but allowed that it would be great for the application researchers and difficult for the architecture researchers. An audience member asked what would be considered "cheating" for benchmarks. Catanzaro replied that the important thing is that the results are reproducible. Several audience members asked for Catanzaro's opinion about several particular benchmark suites. Catanzaro maintained that the conclusions of his presentation set the standard for good comparisons: point of view must be considered and reproducibility is essential, which means there must be sufficient explicit detail provided.

RADBench: A Concurrency Bug Benchmark Suite

Nicholas Jalbert, University of California, Berkeley; Cristiano Pereira and Gilles Pokam, Intel; Koushik Sen, University of California, Berkeley

Nicholas Jalbert presented RADBench, a suite of benchmarks containing 10 concurrency bugs found in large open-source software applications such as Mozilla SpiderMonkey, Apache Web Server, and Google Chromium Browser. Concurrency bugs are plentiful and painful to fix, asserted Jalbert. They are growing ever more commonplace, and they take a long time to diagnose and repair. To facilitate concurrency bug research, RADBench presents concurrency bugs "in the wild" by providing full snapshots of large buggy code and scripts to run the code.

Concurrency bugs have been difficult to track down and mine from old code sources, and tricky to recreate on demand. Inputs and thread scheduling behavior were often insufficient to conjure the bugs. Environmental factors such as interrupts, random numbers generated, and communication socket delays were critical to buggy conditions. Overall, the concurrency bugs did not fit into neat categorizations or classical patterns, making them hard to understand and reproduce. Concurrency bugs break the traditional programming paradigm; they are reactive and wacky.

Deterministic record and replay bug reproduction requires cumbersome overhead. Lightweight bug reproduction would be desirable, but remains an open problem. One approach is to record fewer details and perform analysis offline to reduce online overhead. However, with this technique there is no guarantee of bug reproduction.

Sasha Fedorova (Simon Fraser) asked for clarification of the purpose of RADBench. Jalbert explained that RADBench's purpose is to present a collection of full snapshots of concurrency bugs "in the wild." It is not a bug-finder or debugging tool. Several audience members asked about the scalability of the work done to create RADBench. Jalbert responded that the bugs did not follow classical patterns, so they were difficult to find and hard to categorize. No faster way to identify the bugs was found. Many audience members suggested places to find more concurrency bugs: work done at IBM, code samples from undergraduate class projects, highly commented buggy entries in project code archives. Jalbert agreed that these were all good potential concurrency bug sources. Jalbert stressed that the RADBench work is just the tip of the iceberg, and acknowledged that many problems in addressing and solving concurrency bugs remain.

Day 1, Session 2

Summarized by Jaswanth Sreeram (jaswanth@gatech.edu)

How to Miscompile Programs with "Benign" Data Races

Hans-J. Boehm, HP Laboratories

Several researchers have investigated the distinction between "harmful" data races (so-called destructive races) and "benign" data races, which do not affect the semantics of a concurrent program and can be safely ignored. In this talk, Hans-J. Boehm argues that even such benign data races, while appearing harmless at the program level, can potentially be compiled into code that produces incorrect results.

Data races are considered errors in several language models, including Ada 83, POSIX and C++/C. However, in Java and C#, data races are not considered errors (although the semantics are not clear). One of the problems with benign data races in languages that consider them errors is that the program may work well when compiled with a specific compiler version but, since the language standard prohibits races, a future version of the compiler may produce incorrect code for the same program.

An important work in PLDI '07 on benign data races identified five types of races at the source-code level. Hans described how each of these five examples could be miscompiled by a reasonable compiler to buggy code. In one case, code hoisting by the compiler resulted in a write operation failing to become visible to other threads. Another type of common benign race is when the reader does not care if it sees the old value before the write or the new value after the write. The problem with this benign race is that it is possible for the reader to see a value that is neither the old value nor the new value. If, for example, the writer updates the high bits of the variable and then the low bits in two distinct operations that are separately atomic, then the reader may see the intervening state of the variable.

Hans gave a seemingly innocuous example of a benign race between two threads, each writing the same value to the same variable. Surprisingly, even this race between two redundant writes can be compiled incorrectly. Briefly, this problem is caused by the compiler promoting the shared variable to a register and then both threads nullifying each other's updates, with the outcome that neither write is seen. Hans remarked that spurious self-assignment instructions, which are another factor contributing to this miscompilation, are disallowed in both the POSIX and the upcoming language standards. Burton Smith from Microsoft noted that self-assignments are a common occurrence in some SIMD codes, and this phenomenon may be prevalent in those programs.

Phil Howard (Portland State) asked how many of the benign races described in the PLDI '07 paper can be miscompiled. Hans replied, all of them. Todd Mytkowicz (Microsoft) said there is a paper in the upcoming PLDI that proposes code motion only on data that is thread local with a slowdown of about 20%. He asked if benign races are important and if race-freedom could be enforced strictly. Hans replied that even if that were possible, a programming model that only permits sequential consistency would not be very useful.

Bryan Ford (Yale) noted that programmers will continue to use benign data races even if there are no guarantees of portability or correctness on a different compiler or platform, and he wondered whether there was a way to write racy programs that would not be miscompiled. Hans answered that in C/C++ there are ways to write racy stores in a manner that aligned with the language standards and with low overheads. Bartosz Milewski (Corensic) remarked that weak-atomics in these languages are like benign races that have been sanctified by the standard and hence are okay to use.

Deterministic OpenMP for Race-Free Parallelism

Amittai Aviram and Bryan Ford, Yale University

Determinism in parallel programs has received a considerable amount of attention in recent years. Deterministic concurrency essentially guarantees that a concurrent program will always produce the same output for a given input. This determinism makes concurrency bugs and transient bugs reproducible, thereby easing debugging. It also enables several mechanisms for fault-tolerance that rely on reproducible replaying of computation on a fault in a particular module performing that computation. Finally, determinism helps in the end-to-end verifiability of concurrent programs.

Synchronization primitives are divided into two classes. Naturally deterministic primitives are those in which program logic alone determines which threads are involved and where the synchronization occurs in each thread's execution. The rest can be classified as naturally non-deterministic.

Amittai Aviram presented a form of Deterministic Open MP (DOMP) that has a pure naturally deterministic programming model and race-free semantics. DOMP features the subset of the OpenMP parallel constructs that are deterministic, such as "parallel," "loop," and "sections." Amittai reported that in an analysis of the SPLASH, PARSEC, and NPB suites, around 92% of the occurrences of naturally non-deterministic constructs were to express programming idioms that were purely naturally deterministic at a high level. One of the reasons for this is that OpenMP's deterministic constructs are not expressive enough. For example, the OpenMP "reduction" clause constrains the input type to a scalar and the operation to simple arithmetic or logical operators. OpenMP also lacks a high-level "pipeline" construct, necessitating having to build it from spin-loops and the non-deterministic "flush" construct. The abstractions in DOMP are designed to be expressive enough that programmers do not have to resort to using lower-level naturally non-deterministic constructs to implement them. DOMP offers a generalized reduction clause and a pipeline construct (both of which are naturally deterministic). However, DOMP excludes the naturally non-deterministic constructs from OpenMP, such as "atomic," "critical," and "flush," citing the above observation that these constructs are usually used as low-level components of higher-level idioms which the programming model does not itself provide.

Finally, the DOMP runtime is designed to be race-free, since determinism at the program level requires race-freedom in both the program and the runtime. DOMP uses a "working-copies" programming model which eliminates races. In this model each thread makes a private copy of the shared state and operates on it in isolation. When the thread is finished the runtime merges this updated shared state with the parent thread's pristine copy of the state. If at this point the runtime detects that two or more threads have concurrently modified the same state, then it signals a runtime error.

Steve Johnson (Wave Semiconductor) asked if the merging of copies was data-dependent. Amittai answered that entire regions of data that are in scope have to be merged. Gilles Pokam (Intel) asked how the merging was done and if the order in which states were merged was important. Amittai replied that the runtime simply checks to see if the value in a shared location in some thread's private copy differs from the parent thread's pristine copy; if two or more threads have modified this location, the runtime signals an error. Bryan Ford asked if the runtime's model was similar to snapshot isolation. Amittai replied that it was close to it. Steve Johnson asked, if the runtime signals two writes to the same location as an error, then can the program have runtime errors that are data-dependent and, if so, would that affect the determinism guarantee? Amittai replied that such errors are possible and that the usual testing and quality assurance processes were still required. Hans Boehm (HP Labs) remarked that determinism was in the eye of the beholder and asked whether the authors considered malloc to be deterministic. Amittai replied that malloc uses locks so he wouldn't consider it deterministic.

Day 1, Session 3

Summarized by Bryan Catanzaro (bcatanzaro@acm.org)

Parkour: Parallel Speedup Estimates for Serial Programs

Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor; University of California, San Diego

Before embarking on a project to parallelize an application, it's natural to ask what payoff you should expect from parallelism. Some applications are naturally more parallel than others, and you would like to be confident that your application is parallelizable before actually rewriting it to take advantage of parallelism. Donghwan Jeon presented a method for doing that. Parkour instruments a binary during compilation, and then examines execution traces of the instrumented binary for parallelism, given a model of the target parallel hardware platform.

Parkour uses a variation of critical path analysis (CPA) to estimate parallelism. CPA constructs a dataflow graph of the instructions in a program to discover the dependencies in the program execution. The longest dependency chain in the execution trace is used to find the span, and the overall graph shows the amount of work in the execution trace. Simplistically, CPA can give a parallelism estimate of work/span, but this is only a theoretical limit and is very poorly correlated with realizable parallelism. Parkour uses a hierarchical critical path analysis (HCPA), where a hierarchy is imposed by programmer-visible structures in the original code. Given a model of the type of parallelism a programmer would use to target a particular platform, HCPA then performs local critical path analysis at each node in the hierarchy. Heuristics are used to create parallelism estimates for each level, limiting the parallelism described by the CPA at each level by the realizable parallelism presented by the target platform. Results were presented for two platforms—a multicore x86 platform and the MIT Raw processor—on a selection of benchmarks, compared to hand-parallelized implementations. In general, it was clear that the HCPA approach advocated here gave a fairly accurate estimate of achievable parallelism. One next step would be to use this tool to aid in actually parallelizing an application, since it seems to identify portions of the program which can be parallelized.

Sasha Fedorova asked if Donghwan and his co-authors were applying CPA dynamically as opposed to statically. Donghwan said they were. Mike McCool asked if loop unrolling stole loop-level parallelism, and Donghwan again answered yes. Someone asked how Parkour avoids underestimating speedup. Donghwan replied that they need to use a machine model. Sasha asked if Parkour could be extended to implement parallelism, and Bryan Catanzaro followed up by asking if they inserted OpenMP pragmas. Donghwan said that Parkour does find important regions to parallelize, and they covered that in another paper, but it does not insert OpenMP constructs, as that was too complicated. Craig Mustard (Simon Fraser) asked if Donghwan could elaborate on the planner. Donghwan replied that they roughly model the characteristics of two planners.

Enabling Multiple Accelerator Acceleration for Java/OpenMP

Ronald Veldema, Thorsten Blass, and Michael Philippsen, University of Erlangen-Nuremberg

Ronald Veldema explained that this project is aimed at heterogeneous clusters, including both traditional CPUs and accelerators, in this case GPUs. Since clusters are dynamically loaded, it's difficult to know statically what mixture of traditional CPUs and accelerators an application will have at its disposal during execution on a shared cluster. To make this easier, this work proposed writing parallel platform-independent code, using OpenMP directives embedded as comments in Java source code. When a program is instantiated across a cluster, a modified Java class loader examines the compute resources of each node and then dynamically compiles the computation to fit the resources discovered. Work is dispatched to traditional CPUs using parallel Java execution and to GPUs using CUDA. Work is partitioned automatically by running micro-benchmarks which give an estimate of the capabilities of each of the processors; more capable processors are assigned proportionally more work.

An important part of this project was the memory model used to simplify cluster programming: if the compiler can prove that all references to a particular array are local with respect to the loop being parallelized across cluster nodes, the compiler is able to statically partition the data structure, allowing the program to work with large data structures that cannot fit in a single node's memory. Otherwise, the data structure is duplicated across the cluster. Duplicated data structures are made coherent at OpenMP boundaries in the original program, by keeping a shadow copy of the duplicated data structure and diffing it against the potentially mutated copy created during program execution. Diffs are interchanged to synchronize data across the cluster, all without programmer intervention.

Scaling results from this approach seemed good on the examples they presented, with a clear benefit from using the GPUs to accelerate large computations on the cluster. However, this approach does not allow programmers to take advantage of the widely divergent memory subsystems of the CPU and GPU and, instead, requires programmers to write simple OpenMP parallel loops. The goal of this project, therefore, is not to obtain performance competitive with hand-tuned parallel programs but to enable programmers to very productively exploit heterogeneous clusters, including both CPUs and other accelerators, such as GPUs.

Bryan Catanzaro said that with CUDA 4, you can share GPU memory. Ronald responded that you cannot do MPI message passing between GPUs. Bryan next asked if this can be made deterministic. Ronald answered that they have no control over which hardware will be used—for example, if a GPU uses a different type of float. Bryan countered that Java uses a strict float model, but Ronald responded that this is true only if something is marked as strict. John Kubiatowicz (UC Berkeley) said that this reminds him of work in the '90s, where there were interface issues and communication that didn't quite work. He suggested looking at the older literature to see where it would fit into this work. Ronald said that if he could get MPI to work on a GPU, he would be happy. John pointed out that diffs work well in hardware already, and Ronald replied that we don't need hardware support for finding difference between arrays, as the cost is now negligible.

Day 1, Session 4

Summarized by Sean Halle (seanhalle@yahoo.com)

CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications:

H. Cook, E. Gonina, and S. Kamil, University of California, Berkeley; G. Friedland, International Computer Science Institute; D. Patterson and A. Fox, University of California, Berkeley

Selective Embedded Just-in-Time Specialization (SEJITS) is a framework for applying specializing high-productivity languages to specific hardware at runtime. The programmers use a convenient productivity language and make calls to the SEJITS library for functions they need. For example, using the Python library's Gaussian Mixture Model, the SEJITS library picks the version of the code that performs best on the particular hardware during the run. These versions were created during library development, coded by hand in an efficiency language like C or C++.

This talk focused on Speaker Diarization for speech recognition. This was coded in Python and calls the SEJITS library to leverage the Gaussian Mixture Model (GMM). This GMM library uses multiple kernels, each optimized to particular data characteristics. The SEJITS library picked the best one. Performance was tested on the GTX285 and 480 GPUs, for a number of different input data sizes. It showed an average 32% improvement by using multiple kernel variants and picking the best one dynamically over the course of the run. For larger input sizes, improvement rose to 75%.

When comparing the hand-coded C++/CUDA version during the first run, using SEJITS added 71% overhead. But the library remembered the results, and subsequent runs were 17% faster than the hand-coded C++/CUDA version.

Someone from MIT asked about debugging with all the extra layers. What happens when there's a bug down inside the kernels in the specializer? They're working on the ability to trace where a bug happens; such buried bugs are the same in any multi-level library approach. Is the SEJITS approach the same as just dynamically linking a library? They can do the same thing by linking an appropriately tuned library implementation into Python, so what does SEJITS buy? Armando Fox answered that SEJITS is designed to be used by productivity programmers, not specialists. Also, the specializer can take into account the amount of data at runtime, something you cannot do at compile time. Where is most of the debugging time spent? Most of the development and debugging is in writing the C++ and CUDA low-level kernel code, then embedding that kernel into the SEJITS specializer and linking that to the Python API. Linking to Python is straightforward, with lots of tools available. This linking exposes the CUDA programming model to the framework.

Sasha Fedorova wanted to see a code example, and E. Gonina showed a 40-line example that would be thousands of LOC in C. Sasha then asked how the programmer interacted with SEJITS, and Armando said that the productivity programmer never see SEJITS. Only the kernel expert needs to see the C code. Steve Johnson asked how productivity programmers can improve performance. Gonina answered that they can run the specializer over and over again, while Armando said that they really can't do anything.

Pervasive Parallelism for Managed Runtimes

Albert Noll and Thomas R. Gross, ETH Zurich

Albert Noll summarized the known phenomenon that task scheduling overhead can grow to dominate work as the number of tasks grows large. He showed a graph that marks the critical point where the number of individual tasks is too large relative to the work-time of a single one. He termed this the critical point. Albert emphasized that a JIT is isolated from parallel-task knowledge and so has no means of alleviating this parallel scheduler overhead problem.

Their contribution is to modify the JVM, by modifying the intermediate representation, calling it ParIR. This modification relies on Cilk-style spawn and sync semantics. No mention was made of more interesting semantics to cover a larger class of applications. Noll showed two optimizations that can be done using this IR during the run. The first is merging parallel tasks into a single composite task. He showed that this improves performance when the number of tasks is above a critical point for that task-type. He suggested that profiling information can be collected, and the code recompiled when it discovers that the task size is too small or too large. The JIT modifies the intermediate representation of the code to inline a chosen number of tasks, then it recompiles.

The second optimization done with ParIR is moving invariant code out of a parallel section. . He said that recompilation based on profile information is only possible with a JIT.

Burton Smith commented that the semantics look Cilk-like, which lets the compiler merge iteratively generated tasks easily. However, merging for recursively generated tasks is harder for a compiler inlining approach. Noll agreed. Sean Halle asked if profiling has been implemented—which watches and then does the recompilation? How does it know the critical task size? It's a work in progress. They expect to use hardware counters to collect profiling. Hans Boehm asked how the profiler knows the critical point. It is different for each task-type, and many tasks have variable running time. They will have to perform some kind of statistical analysis. It will have to try many task sizes before it knows whether the current one is critical size.

Poster Session

First set of posters summarized by Shane Mottishaw (smottish@sfu.ca)

Support of Collective Effort Towards Performance Portability

Sean Halle and Albert Cohen, INRIA, France

Sean Halle presented this work prompted by the growing desire to express parallel programs in a single source language and achieve good performance across a range of hardware platforms. The authors recognize that the challenges of productivity, portability, and adoptability cannot be feasibly achieved by any one group, nor can they be solved solely at the language, runtime, or hardware abstraction level. Because there is a wide array of research projects involved in performance portability that involve different runtimes, hardware abstractions, and languages, the authors propose a comprehensive support system which provides a framework in which independent groups can plug their solution (e.g., runtime, language, hardware abstraction) into a layer of the framework, making it available for everyone else to utilize in their own work. There are three main layers: toolchains (languages and compilers), parallel runtimes, and hardware abstractions. This support system is based on Virtualized Master Slave (VMS), the authors' virtualization mechanism, which replaces threads and provides pieces for each level of the support system (e.g., VMS cores for hardware abstraction and plugins for runtimes). To achieve performance, layers of the support system share information with each other. For example, the toolchain can derive information about data and computation needs of a task, which can then be used by the runtime to make scheduling decisions.

Challenges in Real-Time Synchronization

Philippe Stellwag and Wolfgang Schröder-Preikschat, Friedrich-Alexander University Erlangen-Nuremberg

Philippe Stellwag described a parallel NCAS library (rtNCAS) that enables the creation of arbitrary, lock-free, or wait-free data structures. Additionally, the library guarantees that all data structure operations are linearizable. Users of rtNCAS provide a function that implements a (sequential) algorithm to perform an update to a data structure (e.g., an enqueue operation for a FIFO queue). This function returns a structure with the expected old and new values for some state of the data structure and is used by the library to perform an NCAS operation to conditionally swap the old and new values. The rtNCAS library performs the user-defined operation as follows: it first tries to speculatively call the user-defined function and attempts to update the data structure with an NCAS operation. On failure, this operation is delayed by pushing it onto a wait-free FIFO queue (called the operation queue). All threads (regardless of the success of speculative execution) cooperatively execute stalled NCAS operations on the queue. This cooperative behavior combined with speculative execution provides wait-free, disjoint-access parallel access to data structures. The wait-free property also provides upper-bound execution times, which is crucial for real-time applications.

Coding Stencil Computations Using the Pochoir Stencil-Specification Language

Yuan Tang, Rezaul Chowdhury, Chi-Keung Luk, and Charles E. Leiserson, MIT Computer Science and Artificial Intelligence Laboratory

Yuan Tang described a new domain-specific language/compiler framework that allows for efficient stencil computations to be embedded in C++. The user describes their stencil computation (including boundary conditions, shape, dimensionality, data types, and computation kernel) using the Pochoir language. The Pochoir compiler and template library then perform automatic parallelization and cache optimization. Pochoir improves upon a "trapezoidal decomposition" algorithm produced by Matteo Frigo and Volker Strumpen by performing hyperspace cuts to partition n-dimensional grids, yielding more parallelism without sacrificing the cache efficiency of the original trapezoidal decomposition algorithm. The Pochoir runtime system also employs a number of other stencil-specific optimizations. The Pochoir runtime system utilizes Intel Cilk Plus to parallelize code written in the Pochoir language.

Dynamic Prioritization for Parallel Traversal of Irregularly Structured Spatio-Temporal Graphs

Bo Zhang, Duke University; Jingfang Huang, University of North Carolina at Chapel Hill; Nikos P. Pitsianis, Aristotle University; Xiaobai Sun, Duke University

Bo Zhang presented work concerned with the execution of fast/sparse algorithms for all-to-all transformations (e.g., fast Fourier transform, FFT, and fast multipole method, FFM) on multicore architectures. The authors represent FFT or FFM computations as a spatio-temporal directed acyclic graph (ST-DAG) where nodes define computations on spatial entities (e.g., cells in a grid) and edges define spatial and temporal (iteration) dependencies. A parallel traversal of this graph is executed to perform the transformation. At any point in the graph, however, there are often far more tasks available for execution than there are resources to satisfy the requests, and therefore efficient scheduling of these tasks is required. The goal is to first schedule tasks that have the largest impact on the overall execution time. The authors introduce dynamic prioritization to solve this problem. Dynamic prioritization takes into account resource limitations and varying task sizes to perform adaptive ranking of available tasks and executes tasks of higher ranks first in order to reduce the time of the parallel traversal.

Efficient and Correct Transactional Memory Programs Combining Snapshot Isolation and Static Analysis

Ricardo J. Dias, João M. Lourenço, and Nuno M. Preguiça, Universidade Nova de Lisboa, Portugal

Ricardo J. Dias proposed a novel method for reducing memory tracking overhead of transactional memory systems, without sacrificing serializability. Using snapshot isolation (where each transaction executes using a private copy of system state), only write-write conflicts need be detected, thus reducing runtime overhead. However, snapshot isolation may lead to non-serializable executions. To correct this, static analysis (specifically, shape analysis) is used to determine the abstract read and write sets of transactions. These read-write sets can then be compared to determine dependencies between transactions, thus detecting potential conflicts. These conflicts are then corrected automatically prior to executing the transactions. For example, a dummy write can be inserted to force a write-write conflict at runtime.

Second set of posters summarized by Craig Mustard (craiig@gmail.com)

Feasibility of Dynamic Binary Parallelization

Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse, University of Virginia

Jing Yang presented a method of parallelizing binary-only executables by analyzing the execution of the binary and identifying frequently repeated sections which become candidates for parallelization. When the sequential program reaches a point that has previously been traced and is a candidate for parallelization, the sequential execution halts and a parallel version is speculatively executed with a copy of the program state. If the parallel execution fails due to misprediction, then the results are discarded and the sequential version is executed. The authors present a prototype implementation using a simulator that they evaluate with the SPEC2000 and MediaBench benchmark suites. The authors also applied dynamic binary optimization techniques (DBO) to their experiments. The authors find that DBP and DBO enable a 2x speedup for 7 out of 10 floating point benchmarks, and a speedup of 1.27x for integer benchmarks.

Automated Fingerprinting of Performance Pathologies Using Performance Monitoring Units (PMUs)

Wucherl Yoo and Kevin Larson, University of Illinois at Urbana-Champaign; Lee Baugh, Intel Corp.; Sangkyum Kim, Wonsun Ahn, and Roy H. Campbell, University of Illinois at Urbana-Champaign

Wucherl Yoo described a way to automatically fingerprint the performance behavior of applications. The authors first wrote a variety of micro-benchmarks which exhibited different pathological performance characteristics. Then they trained a decision-tree learning algorithm to identify these micro-benchmarks by their exhibited performance characteristics. They then analyzed timesliced profiles of benchmark applications from SPEC and PARSEC and classified particular program phases as exhibiting pathological performance behavior. They achieved an accuracy rate of over 97% for all benchmarks. This work is designed to be added to a profiling suite so that performance characteristics of applications can be classified and instructions can be provided to the user about ways to fix such problems.

PACORA: Performance Aware Convex Optimization for Resource Allocation

Sarah L. Bird, University of California—Berkeley; Burton J. Smith, Microsoft

Sarah Bird and Burton Smith presented PACORA, which endeavors to optimally allocate resources by mathematically modeling the resources to the quality-of-service tradeoff for each program. The authors combine the resource-performance curve of each program and apply convex optimization techniques to find an optimal configuration of resources such that the resources-performance tradeoff for the entire system performance is maximized. In their model, the authors include a special idle process that represents free resources, which contributes to energy savings. Since this optimization can be done iteratively using a gradient descent approach, the authors believe PACORA will be a useful and high performance technique for resource management.

Are Database-style Transactions Right for Modern Parallel Programs?

Jaswanth Sreeram and Santosh Pande, Georgia Institute of Technology

The authors argue that database-style transactions are too rigid to effectively express certain parallel programming patterns. They describe the applications that can benefit from relaxed models as "soft computing applications." Kmeans, for example, benefits from relaxing the guarantees of transactional memory in order to speed up the clustering algorithm by allowing threads to use old and slightly inaccurate values. When the accuracy is allowed to vary by 0.1, the performance increase is significant, while there is no significant increase in error. List search is another example: if a conflict is detected while a thread is searching the list, instead of aborting, the search can be repaired by reading in a new value.

Day 2, Session 1

Summarized by Bryan Catanzaro (bcatanzaro@acm.org)

Balance Principles for Algorithm-Architecture Co-Design

Kent Czechowski, Casey Battaglino, Chris McClanahan, Aparna Chandramowlishwaran, and Richard Vuduc, Georgia Institute of Technology

This paper advocates the use of theoretical modeling to guide architecture development. How much of the architecture should be devoted to cores, for example, and how much to cache? Given a particular parallel architecture, what classes of computation would perform efficiently? This work responds to the observation that simulators are hard to build and require a lot of investment. Once the simulator is built, there are many fundamental assumptions which have been baked into the simulator and are expensive to change. Instead, this paper advocates that processor performance should be theoretically modeled based on high-level characteristics, such as communication and scalability.

The presentation defined a balanced architecture as one where the memory wait time T_mem <= the computation time T_comp. In order to evaluate T_mem and T_comp for a particular algorithm and architecture, one needs to derive a model for both expressions. For T_mem, they used an external memory model (I/O model). For T_comp, they used a Parallel DAG model to discover the work and span of a computation, and then found parallelism using Brent's theorem. The I/O model depends on the parallel cache complexity, which needs to be derived separately from the sequential cache complexity, and depends on scheduling choices. As a punchline, the paper presented results showing that even dense matrix multiplication, the canonically compute-bound kernel, will be bandwidth-bound on GPUs by 2021 if the current scaling trends continue in computation and bandwidth resources. This approach does not consider power dissipation, which might suggeest use of a more general-cost metric.

Crunching Large Graphs with Commodity Processors

Jacob Nelson, Brandon Myers, A.H. Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, and Dan Grossman, University of Washington; Simon Kahan, University of Washington and Pacific Northwest National Laboratory; Mark Oskin, University of Washington

Important graphs found in many real-world applications are both very large and have a low diameter, meaning that they are very hard to partition. This poses complications for graph algorithms which operate on these graphs. The Cray XMT architecture has been very successful at operating on these problems, but it is expensive, and its performance is not competitive on dense problems. This work attempts to utilize insights from the Cray XMT design to enable commodity CPU clusters to perform well on graph algorithms as well as on the denser problems for which they have already been shown to perform well. The main advantages of the Cray XMT over a commodity x86 cluster are the number of contexts the Cray XMT can keep on chip and the high number of outstanding memory transactions it can support. This work describes SoftXMT, a library for x86 processors which aims to provide these advantages to x86 software through the use of lightweight multithreading in software.

SoftXMT uses co-routines to break memory transactions into separate stages. For example, a load becomes a prefetch, a yield, and then a blocking load. The full implementation of SoftXMT will use a compiler which transforms memory transactions into multiple stages. A lightweight library round-robin switches between suspended threads which are waiting on memory requests. The authors presented data which shows that on a single node, the co-routine library used in SoftXMT is efficient, allowing the node to saturate its available memory bandwidth almost as well as the hardware can, as is demonstrated with a simple pointer-chasing benchmark. Future work involves making a complete cluster-based implementation and showing good performance on complete graph algorithm problems.

Day 2, Session 2

Summarized by Amittai Aviram (amittai.aviram@yale.edu)

Multicore Performance Optimization Using Partner Cores

Eric Lau and Jason E Miller, MIT Computer Science and Artificial Intelligence Laboratory; Inseok Choi and Donald Yeung, University of Maryland; Saman Amarasinghe and Anant Agarwal, MIT Computer Science and Artificial Intelligence Laboratory

Increasingly, parallel architecture is exposing more hardware resources to the programmer, who must cope with the contradictory requirements of high performance and energy efficiency in a programming environment whose growing complexity is getting unmanageable. Self-aware programs can manage their resources dynamically, but they burden the CPU with a new meta-program management layer of work, introducing more interrupts and context switches. Eric Lau presented this work which, assuming a tiled general architecture for the future, introduces the idea of partner cores as one possible solution: a smaller core, one-tenth the main core's size and optimized for efficiency, alongside each main core, with its own, lower-powered network router and with dedicated "probes" feeding it performance counters and other status information from the main core. The partner core could then handle the meta-program management, for instance, by running a helper thread to prefetch data from main memory into the cache. Experiments on a simulator showed that partner core memory prefetching could raise performance close to 3x, while more than doubling energy efficiency. Other scenarios that could benefit from partner cores are (1) keeping track of "tainted" pointers in information flow control, (2) running a redundant trailing thread to check output against the main thread periodically for error detection, and (3) prioritizing messages on an event queue.

Phil Howard (Portland State) asked who would program for partner core architectures. Eric answered that it would be system programmers, hopefully, at a level that application programmers would not have to see. Stephen Johnson (Wave Semiconductor) asked whether partner cores could be used to execute assertions so as to provide runtime correctness checks in deployment while staying out of the way of the main program execution. Eric found this very plausible.

Parallel Pattern Detection for Architectural Improvements

Jason A. Poovey, Brian P. Railing, and Thomas M. Conte, Georgia Institute of Technology

Although parallel programming promises performance improvements, optimizing to get the most out of a parallel program poses design challenges. Jason Poovey presented this work on how designing according to parallel patterns can help meet these challenges. Researchers have identified patterns at the level of concept, algorithm, and low-level implementation. The algorithmic level offers both quantifiability and breadth sufficient to help automate optimization and to guide improvements in hardware architecture. For example, thread schedules for pipeline and divide-and-conquer algorithms are quite distinct. While algorithms involving intensive data sharing require the usual MESI (Modified, Exclusive, Shared, or Invalid) cache coherency, algorithms in which data migrate from thread to thread, as in pipelines, could get by with the weaker, and more efficient, MI protocol. Algorithms with little inter-thread communication need far less network bandwidth than those that communicate frequently. Pipeline and divide-and-conquer are two of the six classes that typify parallel algorithms. Those organized by task are task parallel and divide-and-conquer algorithms; those by data, geometric decomposition and recursive algorithms; and those by the flow of data, pipelines and event-based coordination.

In prior work, Jason and colleagues had shown that significant performance improvements are possible when the pattern was known and was used to determine the threadbalancing mechanism. To identify the pattern to which a program belongs, we have two major sources: static detection, including the programmer's own annotations, perhaps through a new API for this purpose; and dynamic detection, which measures data-sharing behavior, thread balance over time, and the uniqueness of instructions to each thread in order to identify signature combinations suggesting particular parallel algorithm patterns. In the case of data sharing, a modest modification to the cache could provide accurate information at a reasonable cost. The team created five benchmarks, one for each pattern except event-based coordination, to serve as reference points ("golden copies"). Once they had compiled measurements for dynamic detection from the reference benchmarks, they ran several standard benchmarks on a simulator and used the same measurements to predict their respective patterns. In each case, they knew from the outset the appropriate pattern, and they found that their data-based predictions had mixed success. They plan further improvements in data gathering methods and modeling in order to improve prediction accuracy.

Michael McCool (Intel) suggested a connection with the previous presentation: could we use partner cores to help detect parallel algorithm patterns? Jason agreed. Another question was how much hardware we need for pattern detection. "Right now, a lot," Jason answered. He then detailed some of the larger sources of data. Hopefully, people may find methods for detecting patterns that will eventually require fewer resources. Could we use software instead of hardware for these measurements? In principle, yes, but it would be even less efficient. Does Jason foresee the need for custom hardware? For thread scheduling, no; but for switching between cache coherency protocols, yes. Is the benefit worth the cost of custom hardware? Switching cache coherency protocols when possible could offer significant performance benefits. Why did the team have to use a simulator? This was the only way that they could collect large data sets for pattern detection. However, their eventual usage model would be dynamic pattern detection and optimizing adjustments during runtime. They also hope to identify more distinct patterns.

Day 2, Session 3

Summarized by Amittai Aviram (amittai.aviram@yale.edu)

Parallel Programming of General-Purpose Programs Using Task-Based Programming Models

Hans Vandierendonck, Ghent University; Polyvios Pratikakis and Dimitrios S. Nikolopoulos, Foundation for Research and Technology—Hellas (FORTH)

Hans Vandierendonck presented this work. Pipelines are a common and essential pattern of parallel programming, in which later tasks depend on earlier ones, sometimes in complex ways, following a directed acyclic graph (DAG) rather than a mere sequence; some tasks are necessarily sequential (such as I/O), while others may be run in parallel but depend on the completion of predecessor tasks. Yet current thread-based programming languages and frameworks, even higher-level ones such as Cilk++ and TBB, are not designed to make pipeline construction easy or intuitive. The awkwardness is evident when the programmer wants to have variables renamed to optimize pipelines. This happens when one task must wait to write to a variable until another task has read the old value (write-after-read, WAR, or anti-dependency), and when one task must overwrite a variable only after another has written a previous value (write-after-write, WAW, or output dependency). Variable renaming enables the second task to proceed without locking or blocking, but Cilk++ and TBB, both thread-based, make the programmer have to program versioning manually with complicated, unintuitive syntax.

A task-based dataflow language could manage the versioning automatically when it infers the need for it, using new extensions to Cilk++ to annotate variable arguments to tasks with their dependency types (indep, outdep, or inoutdep), as well as the standard Cilk++ versioned hyperobject keyword. The result is simpler, more readable code. Such a language could be further extended to accommodate speculative execution, where a thread could execute in parallel while presuming a condition, and then check the state of a variable on which it depends before either committing results or aborting. One could also use dependency-annotated types to prove the deterministic execution of a parallel program.

In implementing task-based dataflow parallelism, the key challenge is to design an efficient scheduler, which could automatically manage dependencies and blocking with a minimum of locks. Hans's team's solution is a ticket queue system, requiring only one lock per task and one on the global queue. Experiments on the SPEC2 benchmarks bzip2 and hmmer show that the task-based dataflow extensions to Cilk++ impose essentially no additional runtime cost relative to standard Cilk++ implementations.

Michael McCool (Intel) picked up on Hans's comment in passing that his team had to accept compromises in designing the scheduler. Hans clarified that the resulting task graph could have extra leaves corresponding to tasks that were waiting to execute. Leo Meyerovich (UC Berkeley) asked whether they had tried comparing their system to task-parallel systems on established benchmarks. Hans said they had compared it with SMPSS, and found that their system performed about the same as SMPSS on Choleski and better than SMPSS on Jacobi, because of the scheduler's reduced overhead.

Parallel Programming with Inductive Synthesis

Shaon Barman, Rastislav Bodik, Sagar Jain, Yewen Pu, Saurabh Srivastava, and Nicholas Tung, University of California, Berkeley

Ras Bodik presented this work. In scientific computing, high-level code synthesizers and libraries ease high-performance code implementation, but only apply to the restricted domains the knowledge of which they reflect. Absent such domain theory, the programmer must use more general, lower-level compilers, with lesser performance, and may need to optimize by hand, which takes time and invites errors, especially when optimization involves parallelizing.

This team's project aims to resolve this dilemma with a tool that enables programmers to specify enough information so that a code synthesizer can do the rest, without the programmer having to know all about the domain. Their solution is based on the SKETCH inductive program synthesis framework, which has the programmer provide (1) a specification of what the equivalent result should be (using a naive algorithm) and (2) a high-level "sketch" or template of what a more efficient algorithm should "look like," in which the programmer uses placeholders ("??") for key constants or variables. The SKETCH synthesizer fills in the placeholders to complete the source code, which can then be compiled down to a high-performance executable. For example, a programmer could provide a specification and template of matrix transposition, and SKETCH would fill in the right index variables in the algorithm to produce a correct and efficient program.

One limitation of SKETCH is that it does not prove correctness, only functional equivalence of its code to the specification for each element in a finite subset of the domain. SKETCH also has scalability limitations, which the current project aims to overcome by experimenting with an interactive refinement and auto-tuning cycle. SKETCH would first produce a naive algorithm based on a simple template (but more efficient than the algorithm in the specification). Next, the programmer would adjust the template based on the resulting algorithm (source code) and resubmit it to SKETCH for automatic tuning. The programmer could continue repeating this cycle. In particular, each phase of refinement could reflect the knowledge of an expert in a distinct domain, each one working at a high level, so that the resulting code would reflect several levels of domain knowledge without requiring any hand optimization. The team applied this technique to the particularly difficult and error-prone task of parallel programming for GPUs, in particular, to solve the maximal independent set (MIS) problem. They began with a sketch that would suggest an exponential algorithm, but refined and auto-tuned to have an efficient dynamic programming algorithm, based on the parallel scan operation.

Another refinement, reflecting "parallel algorithm expert" knowledge, led to an efficient SIMD algorithm to implement the parallel scan network. Finally, refinement according to "GPU tuning expert" knowledge optimized for execution on GPUs by avoiding bank conflicts, having the synthesizer produce an array index translation function so as to map logical arrays to physical arrays. The result was an efficient algorithm, whose further refinement is still in progress.

The ensuing discussion showed lively interest and the need for clarification. SKETCH does not provide a proof of correctness; in the matrix transposition case, the specification might use a matrix of size n where n is small, and then use a small range of larger n to check functional equivalence. At times, the programmer might have to prove correctness by hand. SKETCH looks like constraint programming, but the constraints are in the metalanguage, restricting the search space to a manageable size. In the specification and template, you can also place constraints to force optimizations or to filter out inefficient programs. In principle, one could also use SKETCH to build up a whole library of alternative solutions to the same general problem, but SKETCH cannot yet generate variations automatically in a way that makes sense, which would require that it distinguish meaningful from trivial variations.

Day 2, Session 4

No report is available for this session.

A Relativistic Enhancement to Software Transactional Memory

Philip W. Howard and Jonathan Walpole, Portland State University

Quarantine: Fault Tolerance for Concurrent Servers with Data-Driven Selective Isolation

Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison